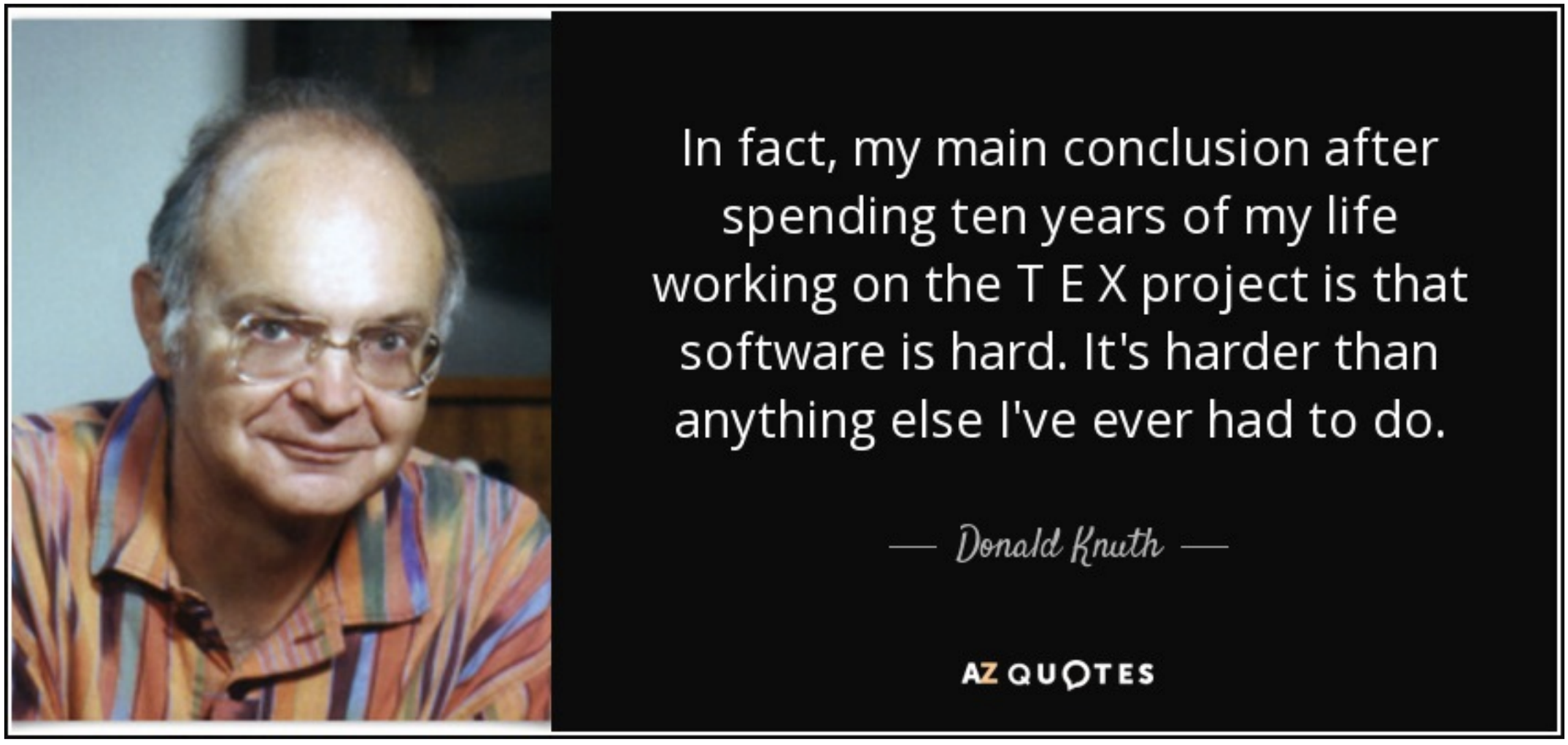
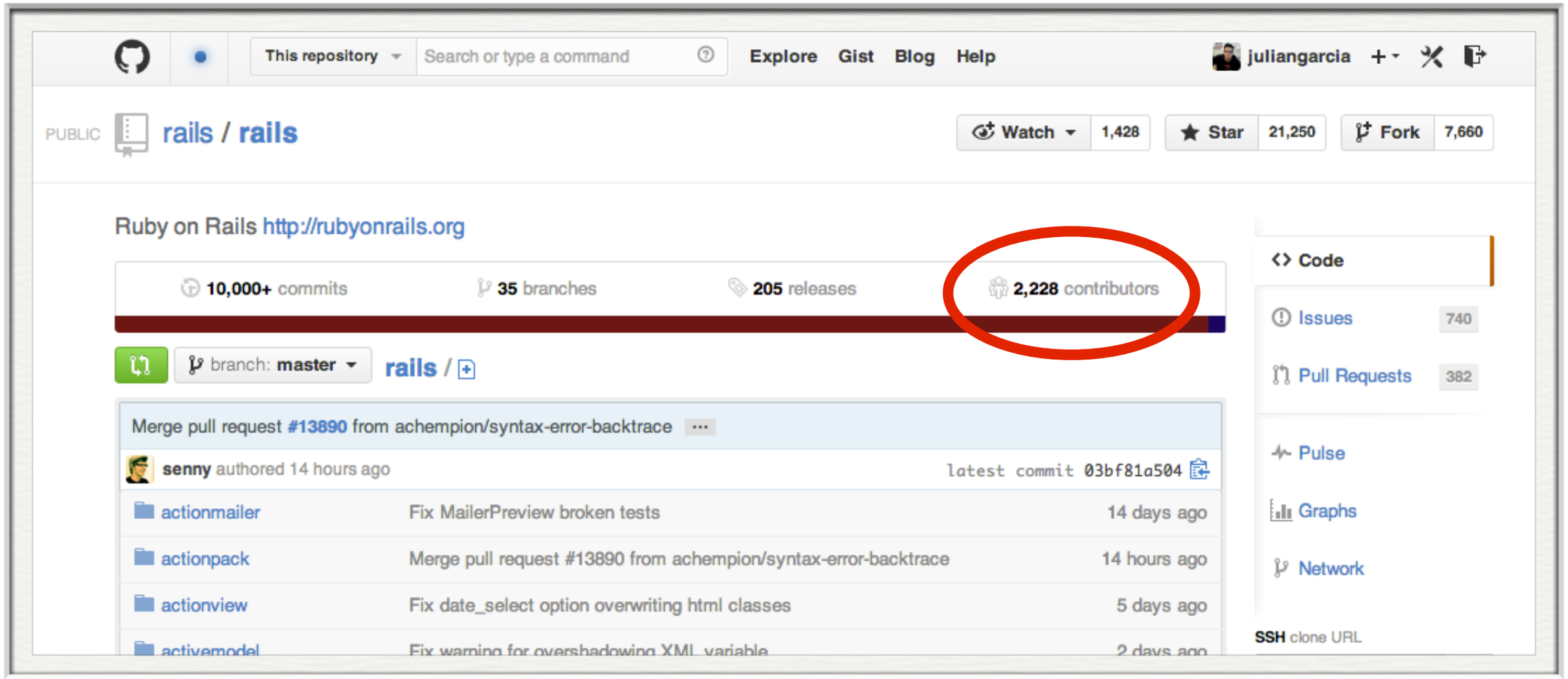
Prepared by Julian García, based on material by
David Albrecht and María García de la Banda

Lecture 13
ADT, Classes and Objects

FIT 1008
Introduction to Computer Science


In fact, my main conclusion after spending ten years of my life working on the T E X project is that software is hard. It's harder than anything else I've ever had to do.
— Donald Knuth —
AZ QUOTES



2228 contributors

Abstraction

Concepts & Ideas

Algorithms

Data Structures

Organise information

Known examples:
Graphs, Tables,
Bitstrings, Lists, Etc.

Manipulation requires we know specifics.

Abstract Data Types

Data structures
along with their operations

Manipulation through operations.

Designed based on: **Abstraction Encapsulation, Information Hiding.**

Computer Programs

Data Types

Tangible things.

Data Structures

Organise information

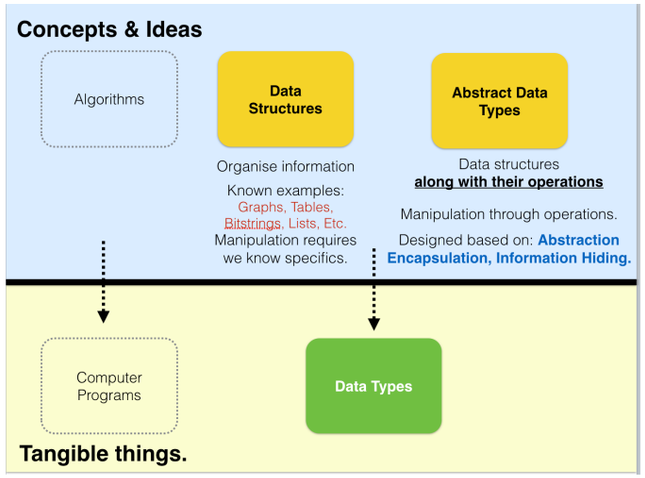
Known examples:
Graphs, Tables,
Bitstrings, Lists, Etc.
Manipulation requires
we know specifics.

Abstract Data Types

Data structures
along with their operations

Manipulation through **operations**.

Designed based on: **Abstraction Encapsulation, Information Hiding.**



Data Types

- Refers to a classification that determines:
  - The possible **values** for that type
  - The **meaning** of those values
  - The **operations** that can be done on them
  - The way those values are **implemented**

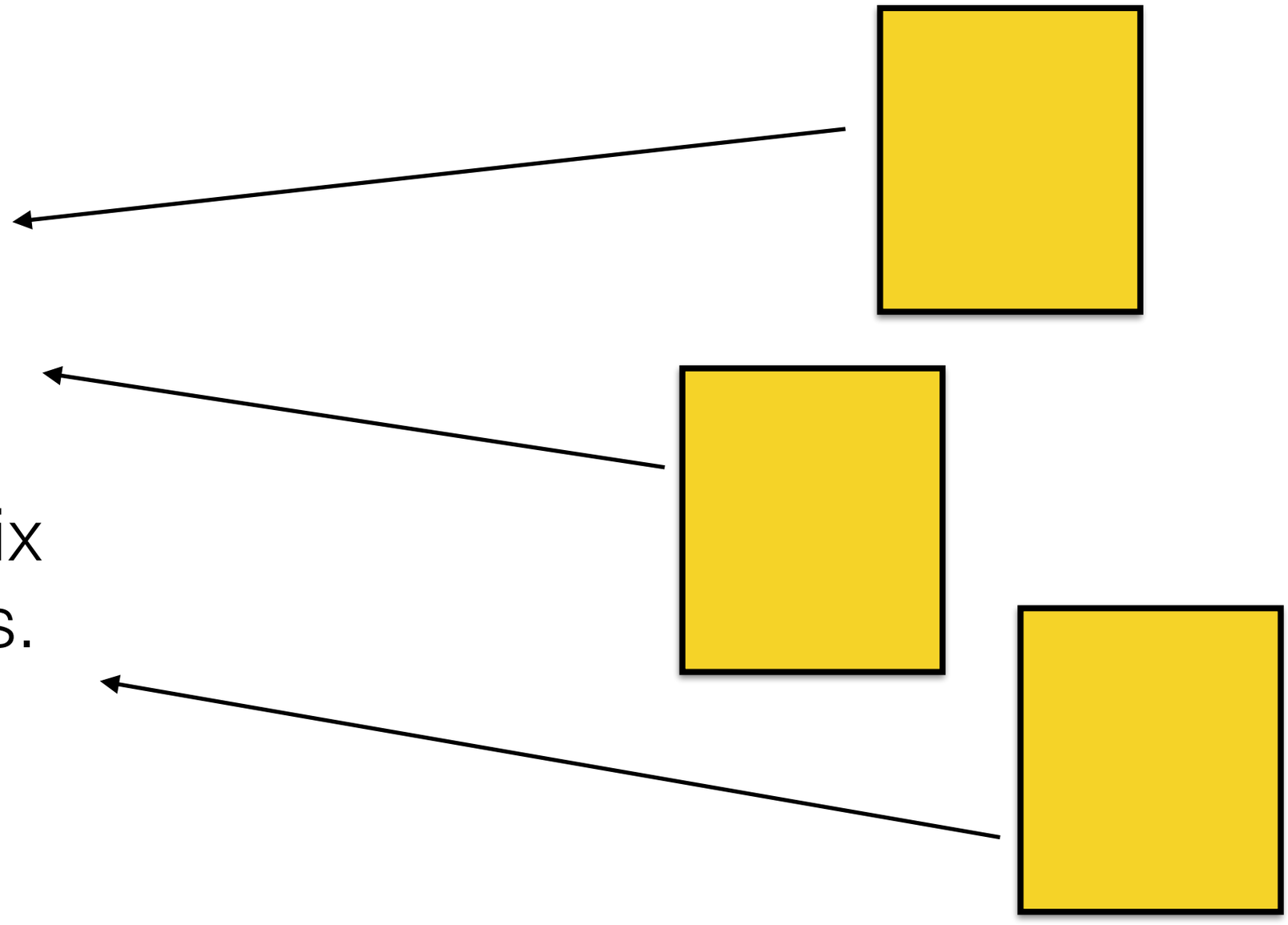
- **Example**: In Java if variable has type ***int***
  - It can take values from -2,147,483,648 to 2,147,483,647
  - Its **meaning** is that of an integer number
  - Can be used in all integer operations (add, subtract, etc)
  - Implemented using 32 bits and specific bytecode operations

used in different parts of a program

Graph

	V_1	V_2	V_3	V_4
V_1	0	1	0	1
V_2	1	0	1	1
V_3	0	1	0	0
V_4	1	1	0	0

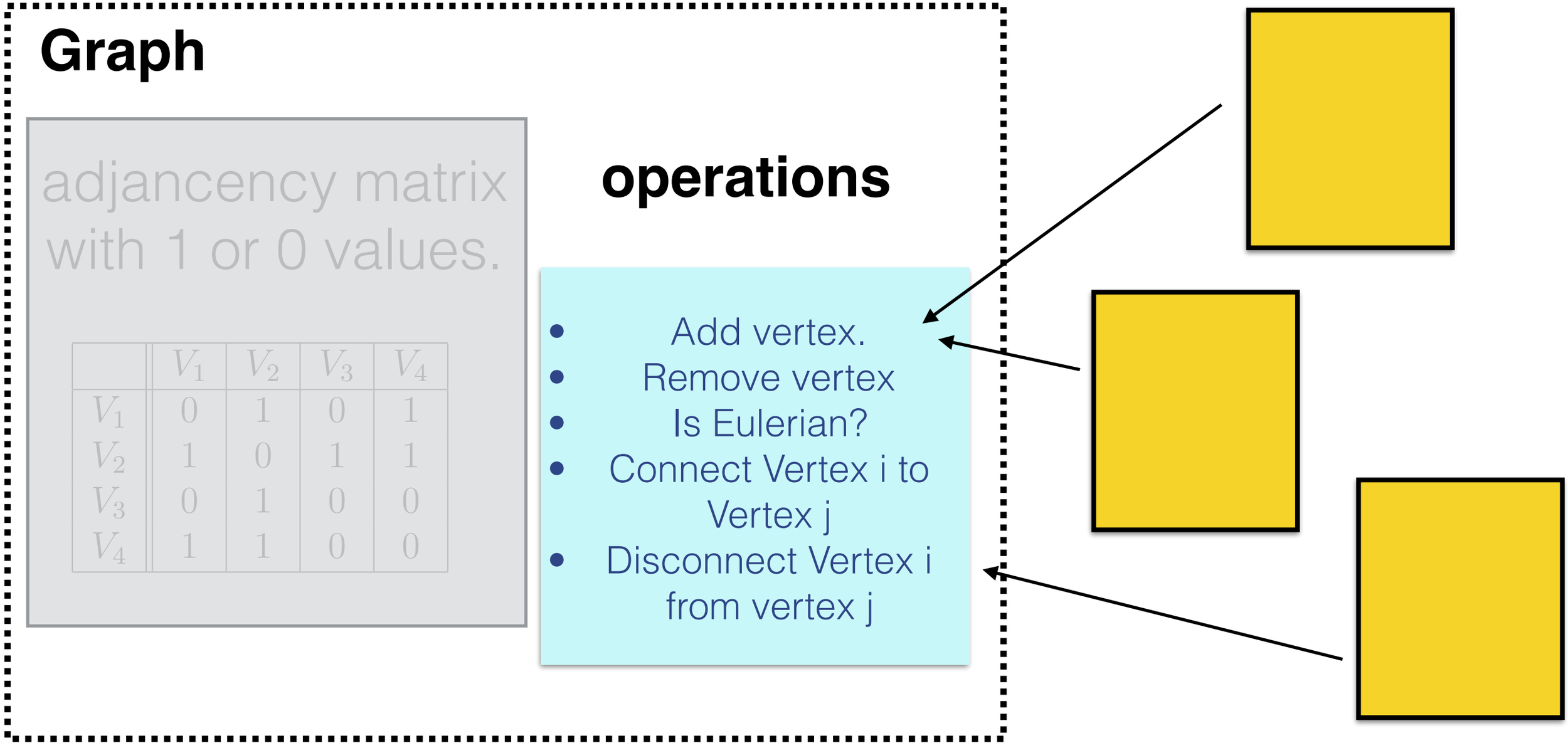
adjancency matrix with 1 or 0 values.



usage requires that I know **how** my graphs are implemented

as an **Abstract Data Type**

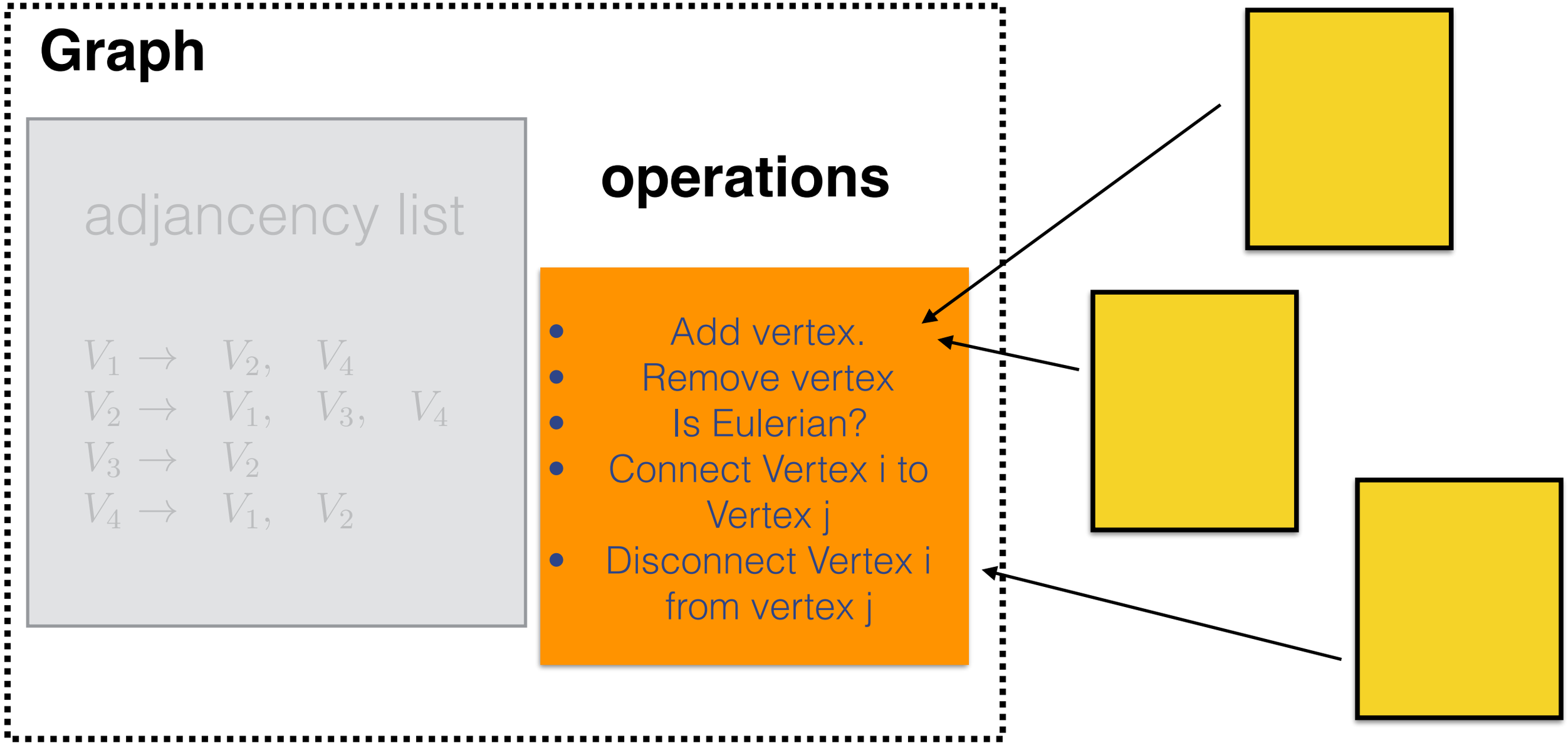
used in different parts of a program



usage through **operations (a.k.a interface)**!

as an **Abstract Data Type**

used in different parts of a program



usage through **operations (a.k.a interface)**!

No changes outside!

Abstract Data Types (ADTs)

- Often **no need to know** how types are **implemented**
- An abstract data type:
  - Provides information regarding:
    - The possible **values of the type** and their meaning
    - The **operations** that can done on them
  - BUT not on its implementation, i.e. how:
    - The values are stored
    - The operations are implemented
  - **Users interact with the data only through the provided operations**

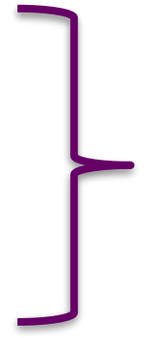
Data Structures

- At some point we must give ADTs an implementation. Information needs to be organised.

- Some ADTs contain several data fields
  - How do we organise the data? How do we access it?

- That is what a **data structure** provides:
  - A particular way in which data is physically organised (so that certain operation can be performed efficiently)

- Example: the **array** data structure
  - Fixed size
  - Data items are stored sequentially
  - Each item occupies the same amount of space



Rational ADT

- **Representation** of rationals
  (two integers, p and q, q not zero).

DATA STRUCTURE

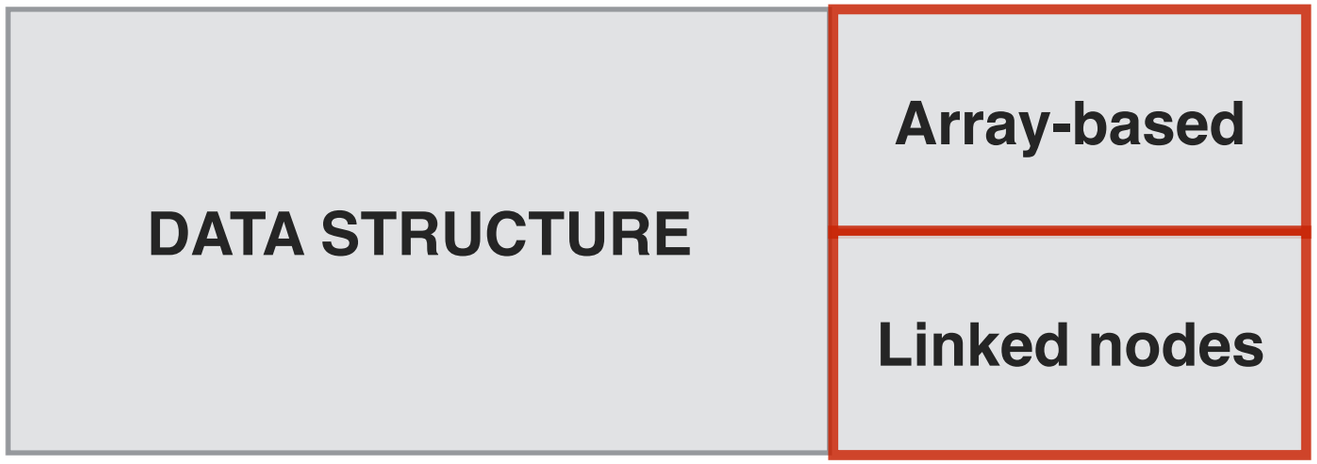
- Possible **Operations**:
  - Arithmetic operations, +, -, /, *
  - Comparison operations, <, >, ==, <=, >=
  - Equality =
  - Read
  - Print

OPERATIONS

Container ADTs

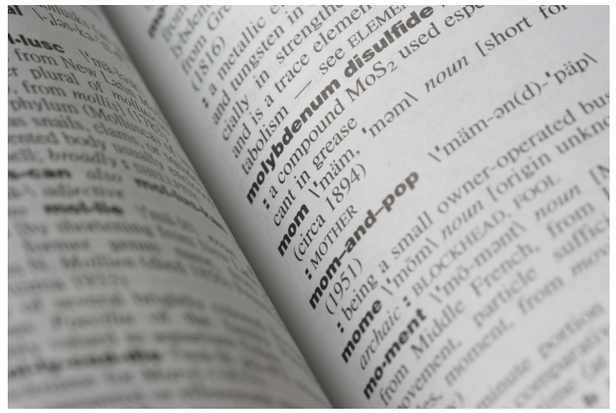
- **Stores** and removes items **independent of contents.**

- **Examples** include:
  - List ADT
  - Stack ADT
  - Queue ADT.



- Core **operations**:
  - ➡ add item
  - ➡ remove item

Dictionary ADT



- Permits **access to data items by content**, e.g., a key.
- Operations
  - Search
  - Insert
  - Delete

OPERATIONS

DATA STRUCTURE	Hash-table
	Binary Search Tree

Example: A dictionary of student names, whose key is the student ID.

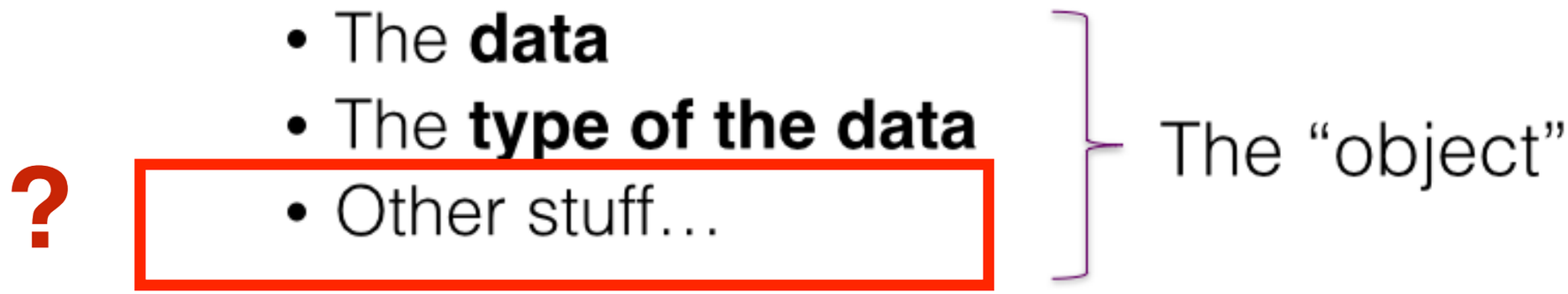
How to implement ADT?

- Learn some **Object Oriented Programming** (in Python)
- In particular, to learn:
  - How to **define basic <u>classes</u>**
  - How to **instantiate them into <u>objects</u>**
  - How to define **<u>methods</u>** and how to use them
  - An example of classes for implement stacks
  - The importance of **namespaces and scoping rules**

- We will **NOT** learn about a major OO component: **inheritance**
  - Not needed for FIT1008
  - Central to the idea of OO

Objects

- **Objects**: blocks of memory containing some data.



- Objects are more than **data** fields (or data attributes).
  They **also** have **methods** that can be performed by the object.

- Like real life: **Objects** interact with each other (through methods.)

Using Objects

- Every value is an object with **data and methods.**

- So, how do we access the data and methods of an object? Through the **“dot”** notation:

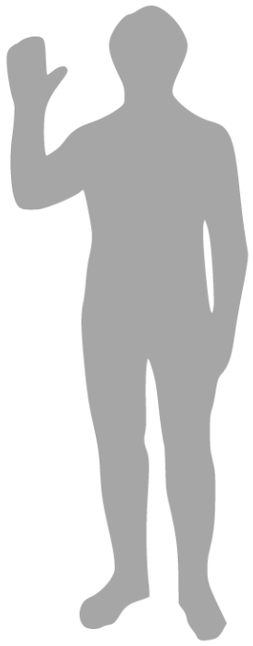
```
>>> "abcd".upper()
'ABCD'
```

```
>>> x = [1,2,3,4]
>>> x.append(5)
>>> x
[1, 2, 3, 4, 5]
```

- Also referred to as “**qualifying**” a variable or method.

- The **dot notation is common** to many languages (e.g. Java.)

Class:



blueprint

Objects:

instances…

Classes

- A class is a **blueprint** for objects, defines **attributes**: data + methods

- Classes are defined using the following syntax:

Convention: Class name starts with uppercase

class is a keyword

class header

```
class ClassName:
    <statement-1>
    .....
    <statement-N>
```

body is indented

class body

Statement-i are mostly “methods”

- Every object is created by **instantiating** a class (see later).

- That is why we say that an **object is an instance of a class.**

Methods

- Define **operations** that can be performed by any object created as an instance of the class
- A method definition looks like a function definition

- Except that:
  - It must appear inside the class (in the body)
  - Its first argument must be a reference to the instance whose method was called
    - By **(a very strong) convention**, this argument is named **`self`**
    - You must **explicitly** add **`self`** to the method **definition**
    - But it is **automatically** added in a **method call**

The __init__ method

double underscore

double underscore

- By **convention** it is the first method in a class.
- First code **executed when creating an instance** (automatically!)
- You can choose not to define it.
- Its first argument (**self**) is a reference to the **instance** whose method was called.

Point Class

don't forget **self**

Local variables

```
class Point:
    def __init__(self, x, y):
        self.x_coordinate = x
        self.y_coordinate = y
```

Instance variables

- **Instance variables** start with **self**:
  - Their **value** is specific to each instance.
  - Their **name** is global to all methods in the class. You can access it from every method in the class.

- If a variable does not start with self it is **local** to the method that binds it. Local variables can't be seen by other methods.

Let's put this into a file called *point.py*

```
class Point:
    def __init__(self, x, y):
        self.x_coordinate = x
        self.y_coordinate = y
```

```
>>> import point
>>> p1 = point.Point(1,3)
>>> p1.x_coordinate
1
>>> p1.y_coordinate
3
>>> p2 = point.Point(-4,7)
>>> p2.x_coordinate
-4
>>> p2.y_coordinate
7
>>> p1.__class__
<class 'point.Point'>
```

Recap

- ADT = data + functionality (together)
- Functions **hide** implementation.
- We implement ADT using classes.

- We create a **class** by:
  - Simply writing class Name, in some file.
  - Adding **indented statements** (such as methods) to it
  - All methods have **self** as first argument

- We create an **object**:
  - By instantiating the class. Calling Name() with the appropriate arguments, as given by **__init__**
  - The object has access to all the **attributes** defined by the class using the **"dot" notation** (e.g., the_stack.push)